Fast Fourier Transform (FFT)

1.1 Background

The FFT is a computationally efficient algorithm for computing the discrete Fourier transform (DFT). The DFT is the *mathematical* entity that is used to perform spectral analysis on samples of a signal. The FFT is the *computational* entity that is used to efficiently compute the DFT of a signal. The DFT of the N signal samples $x(n),\ n = 0, 1, 2, \ldots, N-1$ is defined as

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn} \qquad k = 0, 1, 2, \ldots, N-1,$$

where for notational convenience we have defined $W_N = e^{-j2\pi/N}$. The direct computation of the DFT requires on the order of N^2 additions and multiplies; however, computing the DFT using the FFT algorithm requires on the order of $(N/2)\log_2 N$ additions and multiplies. The "speed improvement factor" of the FFT increases as N gets larger, and is already a whopping 204.8 for N = 1024 and 682.7 for N = 4096. The catch to achieve this speed improvement is that the value of N must be a power of 2[*].

1.2 Description of the FFT Algorithm

1.2.1 Development

The basis of the FFT algorithm lies in the fact that an N–point DFT can be written as the (weighted) sum of two N/2–point DFTs (one DFT of the even-indexed samples and one DFT of the odd-indexed samples) as

$$X(k) = \underbrace{\sum_{n=0}^{(N/2)-1} x(2n) W_{N/2}^{kn}}_{N/2-\text{point DFT}} + W_N^k \underbrace{\sum_{n=0}^{(N/2)-1} x(2n+1) W_{N/2}^{kn}}_{N/2-\text{point DFT}} \qquad k = 0, 1, 2, \ldots, N-1.$$

Each N/2-pt DFTs need only be evaluated for k = 0, 1, 2, … , N/2 –1, because $W_{N/2}^k$ is periodic with period N/2. The complex-valued weighting factor on the second DFT is called a Twiddle Factor (TF). This decomposition of a DFT is shown for N=8 in Figure 1 below; the dots indicate summation (the two values going into a dot get added with the result going to the output) and a twiddle factor beside a line multiplies the value passing through the line. The crossed-line structure that combines the outputs of the two 4-pt DFTs into the 8-pt DFT outputs is called a Butterfly because of its shape. For clarity, one of the butterflies is shown in Figure 2, where the rules for forming the butterfly outputs are given. Note that the twiddle factors in this form of butterfly are W_N^m and $W_N^{(m+N/2)}$.

[*] There are FFT algorithms for which this requirement is relaxed; however, their implementation is considerably more difficult and the so-called radix-two forms (e.g., for length a power of two) are well suited to the current application. In this document we will refer to "the FFT" algorithm to mean the one that is described here, ignoring the fact that many other forms exist.

Some exploitable structure of the twiddle factors of the form $W_N^{(m+N/2)}$ hinge on the properties of complex numbers as shown for N=8 in Figure 3. This allows the butterfly structure to be improved as shown in Figure 4, which is then used in Figure 5 to give an improved form of the DFT decomposition.

This decomposition into half-length DFTs can be done again to each of the two N/2-pt. DFTs, and then again, and then again … until reaching 2-pt DFTs. If the resulting twiddle factors are handled using the "improved form" discussed above, the

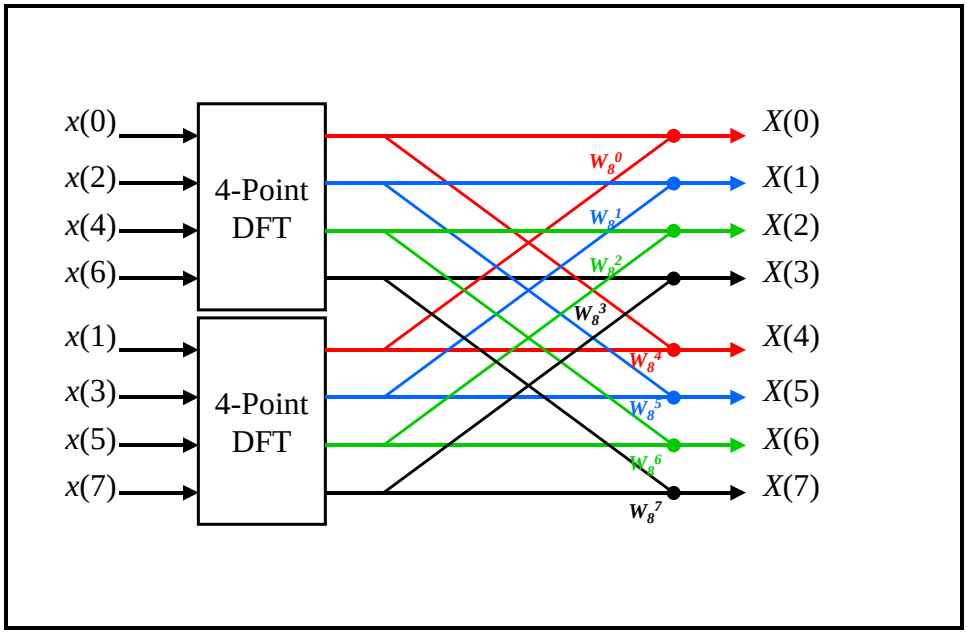

Figure 1: Decomposing 8-pt DFT into 4-pt DFTs

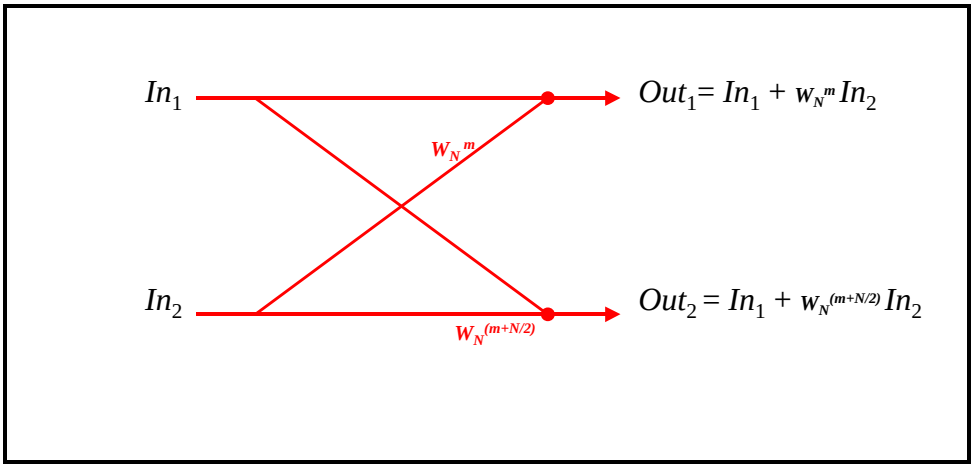

Figure 2: Butterfly Structure

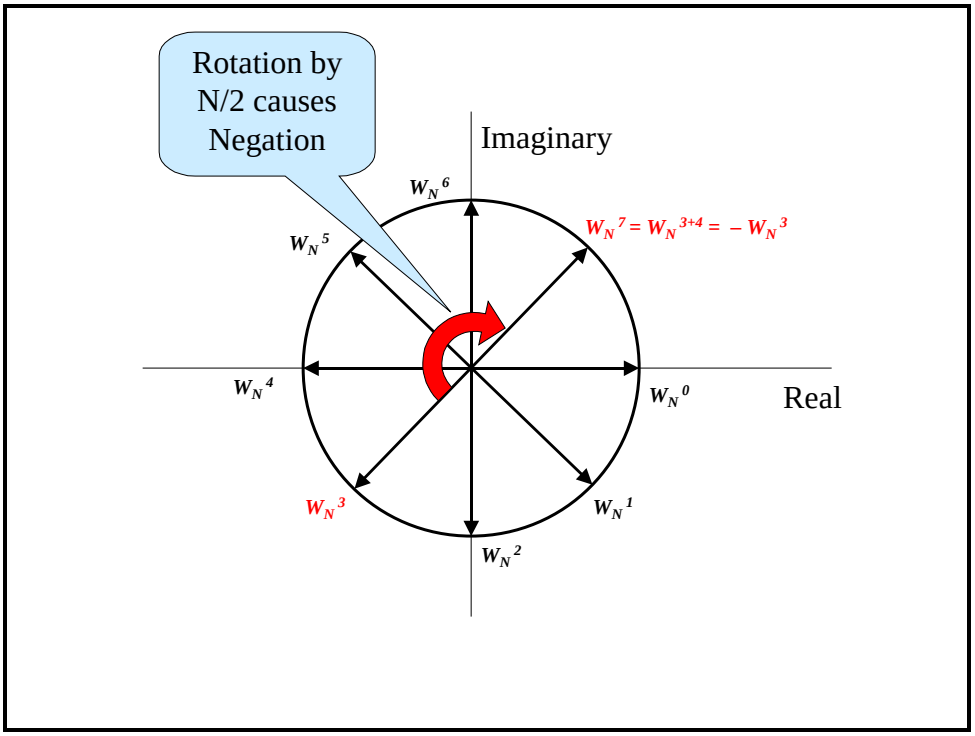

Figure 3: Exploitable Structure of Twiddle Factors

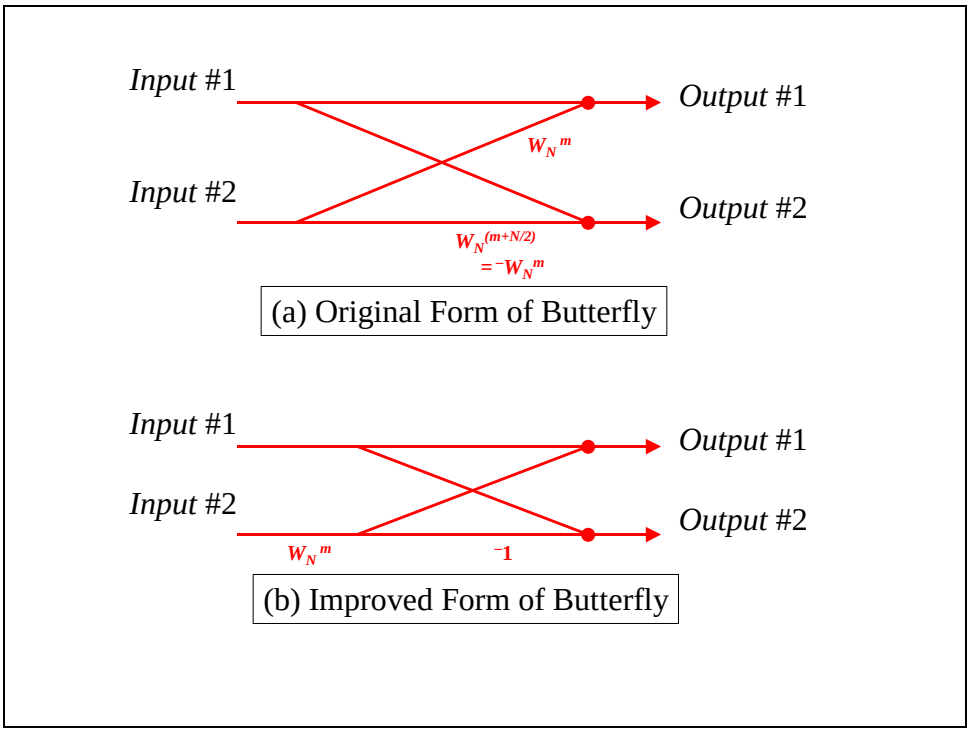

Figure 4: Transition to Improved Form of Butterfly

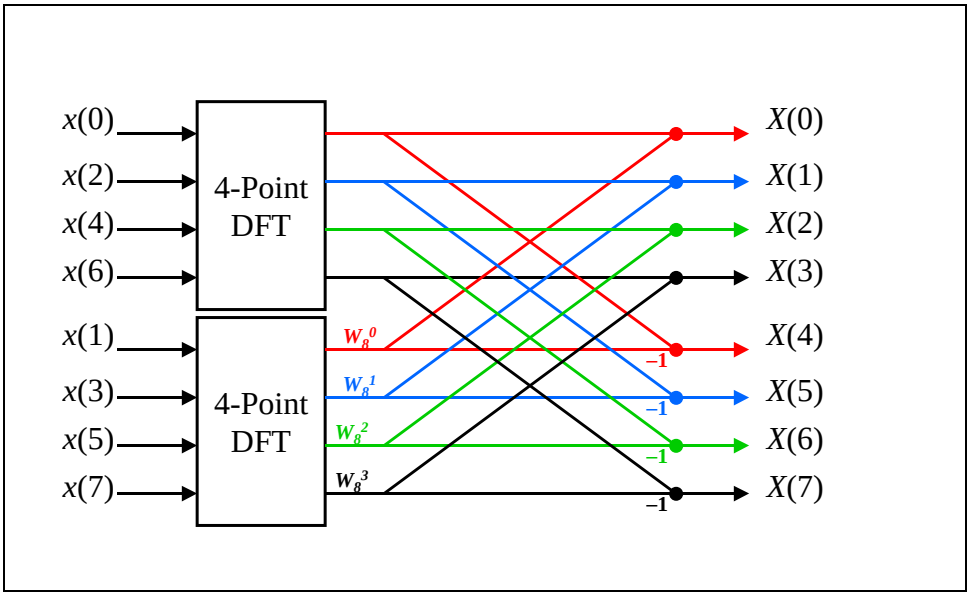

Figure 5: Improved Form of DFT Decomposition

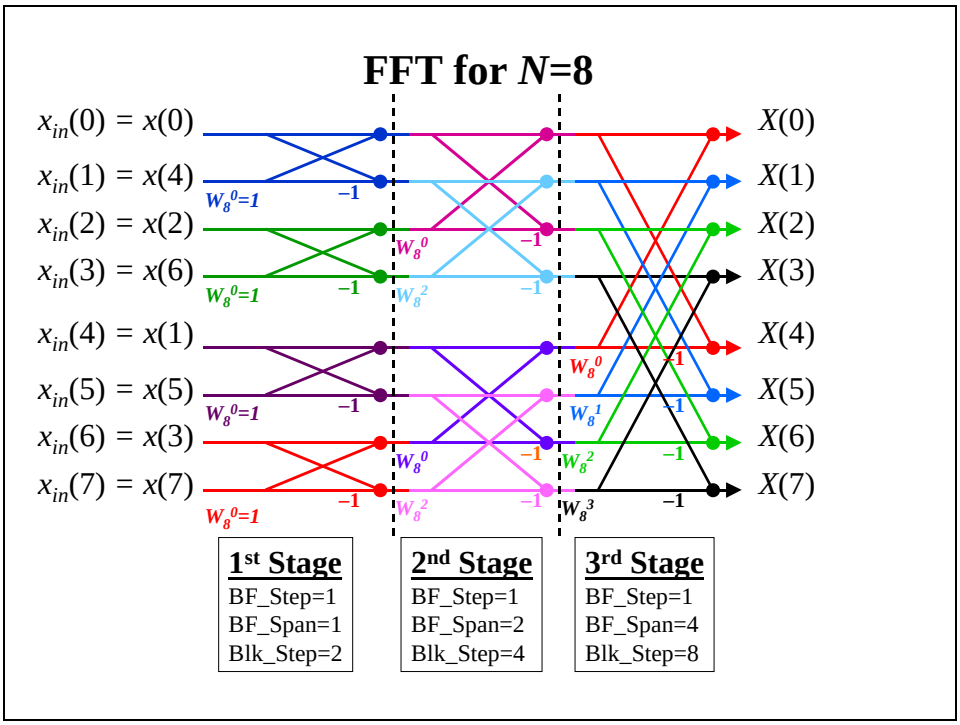

Figure 6: Illustration of FFT Structure for 8-pt DFT

structure that results is the FFT, and is shown for N=8 in Figure 6; the parameters listed at the bottom of Figure 6 will be explained below.

To ensure that the diagram in Figure 6 is understood we present the following walk-through. First note that the diagram is clearly divided into *Stages*, with the output array of a stage being the input of the next stage. Each stage uses $N/2$ butterflies to transform N numbers in its input array into N numbers in its output array, but each stage configures those $N/2$ butterflies differently. Note that it is possible to store the output array of each stage back into the same memory locations that the inputs array was in; this is known as "in-place" computation of the FFT. It is also important to note that the order of the samples in the 1st Stage input array have been reordered by the repeated DFT decompositions discussed above (this reordering will be discussed further below). For indexing purposes we consider x_{in} to be the input array, containing the reordered samples of the signal. The final output array is in sequentially indexed order. Also note that the twiddle factors needed consist of W_N^0, W_N^1, W_N^2, and W_N^3, which is only half the total set displayed in Figure 3; in fact, it is only the "bottom" four of those shown in Figure 3 due to the exploitable structure shown there. The twiddle factors needed for the various stages are shown in Figure 7.

The reordering of the signal samples to form the input array to the first stage follows a very simple rule called bit reversal. To find the proper order for the input samples to be stored in the FFT input array you simply take the ($\log_2 N$)-bit binary representations of the N integers 0 to (N-1) and reverse the order of the bits. The resulting numbers give the needed order of the input signal samples. This procedure is

shown in Figure 8, where it is also indicated that the real effect of the bit reversal is to swap some pairs of signal samples but to leave in place other pairs.

Starting in the 1st Stage at the top butterfly of Figure 6, the diagram says to form the top output of the butterfly by adding $x(4)$ to $x(0)$ and to form the bottom output of the butterfly by subtracting $x(4)$ from $x(0)$. Note that here the twiddle factor in front of the butterfly is 1, as they are for all the 1st Stage butterflies. Likewise, all the other 1st Stage butterflies compute the sums and differences of their inputs and pass the results to their outputs and store them into the memory locations of the inputs. We can view this first stage as having $N/2$ *blocks* with a single butterfly per block (this idea of blocks will become clearer when we discuss the 2nd Stage). Also note that the inputs to each of the 1st stage butterflies are adjacent samples in the (reordered) input array; we say that the 1st Stage butterflies have a *Butterfly Span* = 1. Also, the index distance between 1st Stage blocks is 2, so we define *Block Step* = 2. We also define a parameter called *Butterfly Step* and set it to 1 for the 1st Stage; this will be defined more precisely below for the 2nd and 3rd Stages.

Going now to the 2nd Stage we again see a repetition of the butterfly structure, but with a different structure than in the 1st Stage. Here it is clear that the butterflies have a *Butterfly Span* = 2 (i.e., the inputs to a 2nd Stage butterfly are *two* indices away from each other in the 2nd Stage input array). Although there are $N/2$ butterflies (as in each stage) there are two distinct blocks with two butterflies per block. The first block consists of the top two butterflies and the second block consists of the bottom two butterflies. Within each of these two blocks the index increment between butterflies is 1; that is, the top input of two adjacent butterflies within a block are offset by 1. Thus we say that the *Butterfly Step* = 1. Also, we note that the *Block Step* = 4 in the 2nd Stage; that is, the top input of the bottom block is four indices away from the top input of the top block. The twiddle factors follow the same progression in each of the two blocks. The progression of the twiddle factors within a block is seen to be from W_N^0 to a ¼ of the way around the circle to W_N^2 as seen in Figure 3. That is, the twiddle factor angle step for this is $2\pi/4$. Alternatively, we can consider the set of all the twiddle factors needed for all stages (i.e.,

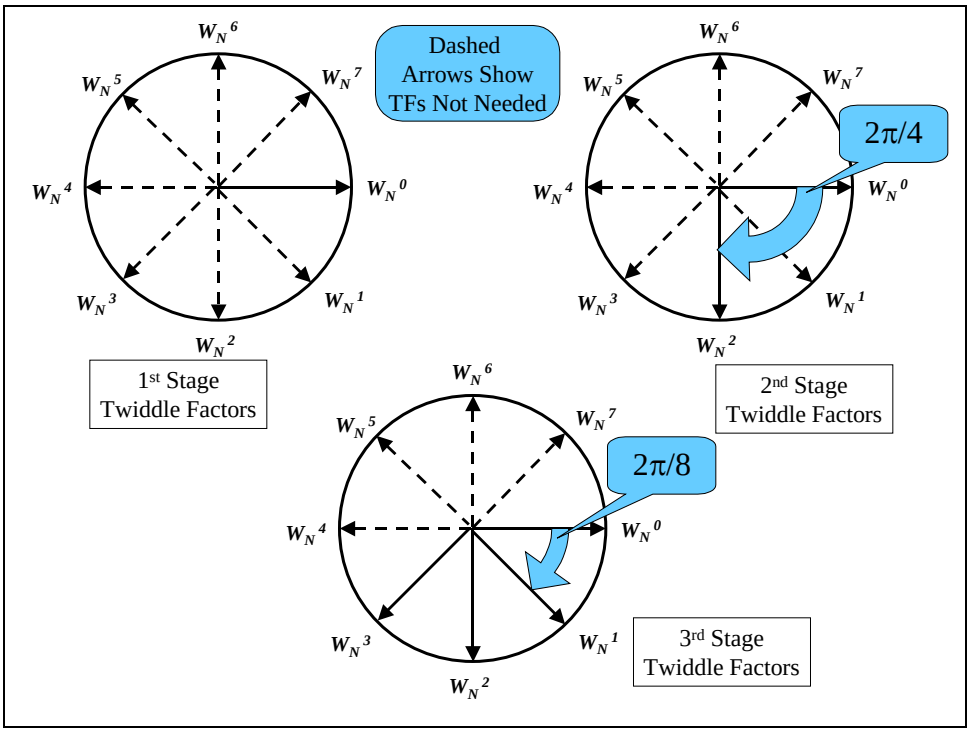

Figure 7: Twiddle Factors for Stages of 8-pt FFT

Figure 8: Bit Reversal Rule Reorders Input Samples

W_N^0, W_N^1, W_N^2, and W_N^3) to be indexed by their powers (0, 1, 2, 3); then the indexes into this set for the two twiddle factors required for this stage is $0\times(N/4)=0$ and $1\times(N/4)=2$. So the twiddle factor index step used in each block of the 2nd Stage is $N/4 = 2$.

Going to the 3rd Stage we yet again see a repetition of the butterfly, but with a different structure than in the 1st or 2nd Stages. We see a single block of $N/2$ butterflies with *Butterfly Step* = 1 and *Butterfly Span* = 4. Because there is only one block the parameter *Block Step* is not really needed; however, it is convenient here to think of *Block Step* = N = 8 (i.e., this value steps beyond the index range of the 3rd Stage input array). The progression of the twiddle factors within the block is W_N^0, W_N^1, W_N^2, and W_N^3 and from Figure 3 we see that twiddle factor angle step for this progression is $2\pi/8$. Alternatively, the indexes into this set for the four twiddle factors required for this stage is $0\times(N/8)=0$, $1\times(N/8)=1$, $2\times(N/8)=2$, and $3\times(N/8)=3$, So the twiddle factor index step used in the single block of the 3rd Stage is $N/8 = 1$.

1.2.2 FFT Implementation Issues

It is clear that there is a lot of structure here that needs to be attended to in the implementation of the FFT. Let's try to use the above characterizations of the N=8 case to extract the general rules for the FFT structure needed for implementation for an arbitrary power-of-two length FFT. We start first with the stage/block/butterfly structure and then discuss the twiddle factor structure. The issue of bit reversal will be handled separately from the FFT routine because for our application the several FFTs of the same

length will be computed in computing the power spectrum; that way the bit reversed indices have to be *computed* only once, but can be used many times for indexing the subsequent FFTs. Thus we assume that the input to the FFT routine is an array of signal samples that have already been reordered according to bit-reversed index values.

Important Note: The indexing schemes given in the descriptions and the pseudo-code below ***DO NOT*** incorporate the indexing needed to handle complex numbers stored as sequential pairs of real and imaginary parts (see Section 1.2.2.5 for more information). Instead they assume that a complex number can be indexed as a single unit rather than a sequential pair of real and imaginary parts. However, the changes needed should be obvious and straightforward.

1.2.2.1 The Structure of Stages, Blocks, and Butterflies

The 1st Stage will always consist of $N/2$ blocks with 1 butterfly/block; each butterfly has *Butterfly Span* = 1; the blocks are offset by 2 and therefore the *Block Step* = 2; by default we say that the *Butterfly Step* = 1. In each subsequent stage we have twice as many blocks, each having half as many butterflies/block; each butterfly has double the span; the block step is twice as large; and the butterfly step stays constant at 1. This continues until there is only one block of $N/2$ butterflies; that is, there are $\log_2(N)$ stages. This structure is captured in Table 1, where the stages have been numbered using an origin-0 scheme.

Table 1: FFT Parameters for Various Stages

Stage →	0	1	2	3	…	$\log_2(N)$-1
Num Blocks	$N/2$	$N/4$	$N/8$	$N/16$	…	$N/N = 1$
BFs/Block	1	2	4	8	…	$N/2$
BF Span	1	2	4	8	…	$N/2$
Block Step	2	4	8	16	…	N
BF Step	1	1	1	1	…	1

From this it is clear that these parameters can be specified in terms of the origin-0 index k of the stage number as shown in Table 2.

Table 2: Rules for *k*th Stage of FFT

	k^{th} Stage (Origin-0)
Num Blocks	$N/2^{(k+1)}$
BFs/Block	2^k
BF Span	2^k
Block Step	$2^{(k+1)}$
BF Step	1

From this discussion we find that the FFT can be implemented using an outer loop over the stages and then at each stage a loop over the blocks and then, finally, for each block a loop over the butterflies in the block. The results in Table 2 are used to get the correct size loops and the correct indexing of the butterflies in the various parts of the loops. Note that all the quantities in Table 2 depend only on the stage index and can therefore be computed just once per stage. At each stage we need a pointer to the beginning of each block, and that is given by multiplying the *Block Step* by the origin-0 block index (Blk_Pntr = m*Blk_Step). Within each block we need a pointer to the

FFT's Stage/Block/Butterfly Structure
(Doesn't Include Bit Reversal or Twiddle Factor)

```
for k = 0 to [log2(N) – 1]                          % Loop over stages
        Num_Blks_Stage = N/2^(k+1)                  % compute # of blocks for current stage
        Num_BFs_Block = 2^k                         % compute # of butterflies/block in current stage
        BF_Span = 2^k                               % compute span of butterflies in current stage
        Blk_Step = 2^(k+1)                          % compute step between blocks in current stage
        BF_Step = 1                                 % define step between BFs in current block
        for m = 0 to Num_Blks_Stage–1               % Loop over blocks in current stage
                Blk_Pntr = m*Blk_Step               % compute pointer to beginning of current block
                for n = 0 to Num_BFs_Block–1        % Loop over butterflies in current block
                        TF=TF_Look_Up(n,k)          % compute TF for current BF (indep. of  block #)
                        Input_1 = Blk_Pntr + n*BF_Step   % compute pointer to first input to BF
                        Input_2 = Input_1 + BF_Span      % compute pointer to second input to BF
                        BF_In_1 = x(Input_1)        % retrieve input #1 to the BF
                        BF_In_2 = x(Input_2)*TF     % retrieve input #2 and apply TF
                        BF_Out_1 = BF_In_1 + BF_In_2     % compute BF output #1
                        BF_Out_2 = BF_In_1 –  BF_In_2    % compute BF output #2
                        x(Input_1)=BF_Out_1         % store BF outputs back in input array
                        x(Input_2)=BF_Out_2
                end on n                            % end of loop over BFs in Block
        end on m                                    % end of loop over Blocks in Stage
end on k                                            % end of loop over Stages
```

current butterfly's two inputs. The top input to the butterfly is indexed by multiplying the *Butterfly Step* (always 1) by the origin-0 butterfly index and adding the result to the *Block Pointer* (Input_1 = Blk_Pntr + n*BF_Step); the bottom input to the butterfly is indexed relative to the top input by adding an offset of *Butterfly Span* (Input_2 = Input_1 + BF_Span). These results are shown in the pseudo-code shown in the box titled "FFT's Stage/Block/Butterfly Structure", which doesn't address the bit reversal or the twiddle factor computation that is needed.

1.2.2.2 Twiddle Factor Computation

For efficiency the twiddle factors should be pre-computed before entering into the routine given above. Then the call "TF_Look_Up" in the pseudo-code above just does a look-up of the needed twiddle factor. In fact, for the application here the TF table can be computed just once for all the FFTs to be computed (i.e., outside the loop over the blocks). We'll first address how to compute the twiddle factors and then discuss how to index the computed values when they are needed in the various blocks.

Computation of the twiddle factors requires computation of sines and cosines, as is seen from Euler's formula, namely

$$\begin{aligned} W_N^m &= e^{-j2\pi m/N} \\ &= \cos(2\pi m/N) - j\sin(2\pi m/N), \end{aligned}$$

where j is the imaginary unit number ($j = \sqrt{-1}$), the first line is by definition of the twiddle factor and the second line is by Euler's formula. Remember that we only need to compute the twiddle factors for half the values of m, namely $m = 0, 1, 2, \ldots, (N/2) - 1$, rather than all the way to $m = N - 1$. These sines and cosines are needed at uniformly spaced angles (i.e., spacing of $\Delta\theta = 2\pi/N$), and that fact can be used to find an efficient way to compute them recursively. This recursion is based on the following trigonometry identities:

$$\begin{aligned} \cos(A+B) &= \cos(A)\cos(B) - \sin(A)\sin(B) \\ \sin(A+B) &= \sin(A)\cos(B) + \cos(A)\sin(B). \end{aligned}$$

This is applied to compute sine and cosine at some multiple of $\Delta\theta$ by recognizing that $m\Delta\theta = (m-1)\Delta\theta + \Delta\theta$ and using the trig identities to write

$$\begin{aligned} \cos(m\Delta\theta) &= \cos((m-1)\Delta\theta)\cos(\Delta\theta) - \sin((m-1)\Delta\theta)\sin(\Delta\theta) \\ \sin(m\Delta\theta) &= \sin((m-1)\Delta\theta)\cos(\Delta\theta) + \cos((m-1)\Delta\theta)\sin(\Delta\theta). \end{aligned}$$

Thus, if we have already computed sine and cosine of $\Delta\theta$, we can use them to define a recursive computation for the sines and cosines needed for the twiddle factors. Defining $S_{\Delta\theta} = \sin(\Delta\theta)$, $C_{\Delta\theta} = \cos(\Delta\theta)$, $C(m) = \cos(m\Delta\theta)$, and $S(m) = \sin(m\Delta\theta)$ we have the recursions between adjacent angles for the twiddle factors

$$\begin{aligned} C(m) &= C_{\Delta\theta}C(m-1) - S_{\Delta\theta}S(m-1) \\ S(m) &= C_{\Delta\theta}S(m-1) + S_{\Delta\theta}C(m-1). \end{aligned}$$

Now note that for $m = 0$ the values of the sine and cosine are known to be

$$\begin{aligned} C(0) &= \cos(0\times\Delta\theta) = 1 \\ S(0) &= \sin(0\times\Delta\theta) = 0, \end{aligned}$$

and can be used to initialize the recursion. Once the values of $C(m)$ and $S(m)$ are computed by the recursion, then the twiddle factors can be computed using Euler's formula as

$$W_N^m = C(m) - jS(m).$$

Thus, the computation of the twiddle factors can be done using the pseudo-code shown below, which requires only two calls to computationally expensive trigonometric subroutines; the rest are computed through the computationally inexpensive recursion.

Pre-Computation of the Twiddle Factors

```
Del_Theta = 2π/N                          % Compute the angle increment
C_Del = cos(Del_Theta)                    % Compute cosine of angle increment; multiplier used in recursion
S_Del = sine(Del_Theta)                   % Compute sine of angle increment; multiplier used in recursion
C(0) = 1                                  % Compute initial cosine to initialize recursion
S(0) = 0                                  % Compute initial sine to initialize recursion
TF_List(0)=1
for m = 1 to (N/2 – 1)                    % Loop over needed TFs; Only need N/2 TFs, not all N of them
        C(m) = C_Del*C(m-1) – S_Del*S(m-1)   % Compute m^th cosine value via the recursion
        S(m) = C_Del*S(m-1)  + S_Del*C(m-1)  % Compute m^th sine value via the recursion
        TF_List(m) = C(m) – j*S(m)           % Compute m^th TF (See below for discussion of complex
#s)
end on m                                  % end of loop over needed TFs
```

1.2.2.3 Twiddle Factor Indexing

As the 8-pt example FFT above showed, the indexing of the twiddle factors from the pre-computed list has a definite structure. This section will describe this structure. Note from Figure 6 that the structure of the twiddle factors is the same for every block in a stage; thus, the twiddle factor indexing structure depends only on the current stage. Also, the number of different twiddle factors needed in each block is equal to the number of butterflies in the block (i.e., one twiddle factor per butterfly). From Figure 7 notice that in each stage, the needed twiddle factors follow a uniform angular progression. Thus, the key to the twiddle factor structure is determining the angular step between twiddle factors as a function of the stage index. Generalizing from Figure 7 we see that the first stage needs a single twiddle factor of $W_N^0 = 1$; the second stage needs two different twiddle factors, namely W_N^0 and $W_N^{N/4}$; the third stage needs four twiddle factors, namely $W_N^{(0\times N/8)} = W_N^0$, $W_N^{(1\times N/8)} = W_N^{N/8}$, $W_N^{(2\times N/8)} = W_N^{N/4}$, and $W_N^{(3\times N/8)} = W_N^{3N/8}$; etc. Thus we see that at the k^{th} (indexed origin-0) stage we need 2^k twiddle factors chosen uniformly spread in angle from the set of all the $N/2$ needed twiddle factors (i.e., W_N^0, W_N^1, W_N^2, …, $W_N^{(N/2-1)}$. Equivalently, we are choosing 2^k equally spaced indices from the set of indices 0, 1, 2, … , $N/2 - 1$; thus, the step between these indices must be *TF Index Step* = $(N/2)/2^k = N/2^{(k+1)}$. Thus, the required indices into the twiddle factor table are $m\times$(*TF Index Step*) = $mN/2^{(k+1)}$ for $m = 0, 1, 2, \ldots, 2^k - 1$.

1.2.2.4 Pseudo-Code for FFT

Notes:

1. This pseudo-code assumes that the input array is already in bit-reversed order (see Section 1.2.3 for details of bit reversal)
2. This pseudo-code does not check that the length of the signal data is a power of two; the actual code produced ***may*** need to do that since this FFT algorithm will ***NOT*** work for signal lengths that are not a power of two.
3. The indexing in this pseudo-code does ***NOT*** handle the fact that complex numbers typically have their real and imaginary parts stored in consecutive memory locations; instead, it assumes that the real and imaginary parts are stored as a single unit. That is, it is assumed that the data and twiddle factors are stored as arrays of complex numbers.

Pseudo-Code for FFT

```
%% Input signal data is in array called Data, in bit-reversed order

%% TF_List has been computed prior to entering this code

%%%%%%%    Stage/Block/Butterfly Loops    %%%%%%
for k = 0 to [log2(N) – 1]                          % Loop over stages
        Num_Blks_Stage = N/2^(k+1)                  % compute # of blocks for current stage
        Num_BFs_Block = 2^k                         % compute # of butterflies/block in current stage
        BF_Span = 2^k                               % compute span of butterflies in current stage
        Blk_Step = 2^(k+1)                          % compute step between blocks in current stage
        BF_Step = 1                                 % define step between BFs in current block
        TF_Index_Step = N/2^(k+1)                   % compute the step between TF indices for this stage
        for m = 0 to Num_Blks_Stage–1               % Loop over blocks in current stage
                Blk_Pntr = m*Blk_Step               % compute pointer to beginning of current block
                for n = 0 to Num_BFs_Block–1        % Loop over butterflies in current block
                        TF_Index = n*TF_Index_Step  % compute the index to the TF for this BF
                        TF=TF_List(TF_Index)        % look-up in the pre-comp. TF_List the TF for this BF
                        Input_1 = Blk_Pntr + n*BF_Step      % compute pointer to first input to BF
                        Input_2 = Input_1 + BF_Span         % compute pointer to second input to BF
                        BF_In_1 = Data(Input_1)             % retrieve input #1 to the BF
                        BF_In_2 = Data(Input_2)*TF          % retrieve input #2 and apply TF
                        BF_Out_1 = BF_In_1 + BF_In_2        % compute BF output #1
                        BF_Out_2 = BF_In_1 –  BF_In_2       % compute BF output #2
                        Data(Input_1)=BF_Out_1              % store BF outputs back in input array
                        Data(Input_2)=BF_Out_2
                end on n                            % end of loop over BFs in Block
        end on m                                    % end of loop over Blocks in Stage
end on k                                            % end of loop over Stages

%% Result of FFT is in the array called Data, in sequential order of the DFT, i.e. X(0), X(1), … X(N-1).
```

Pseudo-Code Box 1: FFT Code (w/o Bit-Reversal Indexing)

1.2.2.5 Storing and Computing with Complex Numbers

Some comments are in order on the fact that the numbers that have to be dealt with in an FFT are complex-valued. The values computed above for the sines and cosines (i.e., $S(m)$ and $C(m)$) are real-valued numbers. The twiddle factors are (in general) complex-valued numbers, in that they have a real part and an imaginary part. In an implementation it is customary to store the real and imaginary parts of complex numbers in adjacent memory locations. Thus, when storing the sequence of the complex-valued twiddle factors $TF(m) = C(m) - j\,S(m)$ for $m = 0, 1, 2, \ldots, N/2 - 1$ it is customary to store them in sequential memory locations as $C(0)$, $-S(0)$, $C(1)$, $-S(1)$, $C(2)$, $-S(2)$, … $C(N/2 - 1)$, $-S(N/2 - 1)$. Likewise, the input signal data is complex-valued and therefore should be stored similarly as alternating between real and imaginary parts. Thus, all multiplications and additions that are specified in these algorithms must be complex additions and multiplications. Complex addition consists simply of adding real part to real part to get the real part of the result and adding imaginary part to imaginary part to get the imaginary part of the result. Complex multiplication consists of the following:

$$(R_1 + jI_1) \times (R_2 + jI_2) = (R_1R_2 - I_1I_2) + j(I_1R_2 + R_1I_2)$$

Thus, for example, if a signal sample's real and imaginary parts are stored in sequential memory locations as X_R and X_I and they must be multiplied by twiddle factors stored in sequential memory locations as $C(m)$ and $-S(m)$, then the product gets stored in sequential memory locations as $[X_R\,C(m) + X_I\,S(m)]$ for the real part of the product and $[X_I\,C(m) - X_R\,S(m)]$ for the imaginary part of the product.

Important Note: The indexing schemes given in the pseudo-code above DO NOT incorporate the indexing needed to handle complex numbers stored as sequential pairs of real and imaginary parts. However, the changes needed should be obvious and straightforward.

1.2.3 Bit Reversal Pseudo-Code

There are many ways to implement the bit-reversed indexing needed for the FFT, and in fact an active area of research today is the development of efficient bit-reversal techniques that are tailored to the memory structures of today's computer architectures. In high-throughput cutting-edge implementations the implementation of the bit reversal part of the FFT algorithm is crucial. However, for the application here it is felt that a "garden variety" method will be suitable. This is believed because (1) the application is not a high-throughput case due to the fact that the FFT result is intended for human viewing not further processing, and, most importantly, (2) the bit-reversed indices are used for computing several FFTs and therefore their overhead is spread over these multiple FFTs, making the minimization of the overhead less critical. Furthermore, without a detailed evaluation of the memory/computer architecture to be used it is impossible to perform a valid tradeoff between the various competing bit-reversal methods.

There are two main steps here: (1) compute the bit-reversed indices, and (2) use those indices to reorder a block of signal data to be applied to the FFT. The basic need for bit reversal was discussed in 1.2.1. The implementation presented here relies on a

simple recursion known as Horner's recursion that allows efficient evaluation of polynomials. Let $k = 0, 1, 2, 3, \ldots, N-1$, where $N=2^n$, be the index of the signal samples and let $\tilde{k}$ be the corresponding bit-reversed index for the data array (the ~ symbol graphically evokes the idea of reversal). Then let the n-bit binary representation of k be $k_{n-1}\, k_{n-2} \ldots k_3\, k_2\, k_1\, k_0$, from MSB to LSB. Then, the n-bit binary representation of $\tilde{k}$ is $k_0\, k_1\, k_2\, k_3 \ldots k_{n-2}\, k_{n-1}$, from MSB to LSB. Thus, we can write

$$\tilde{k} = k_0 2^{n-1} + k_1 2^{n-2} + k_2 2^{n-3} + k_3 2^{n-1} + \cdots + k_{n-2} 2^1 + k_{n-1} 2^0,$$

which can be viewed as a polynomial (with coefficients given by the k_i) evaluated at the value 2. Thus, we can use Horner's recursion to rewrite this in a more computationally efficient way

$$\tilde{k} = k_{n-1} + 2(k_{n-2} + 2(k_{n-3} + \cdots + 2(k_2 + 2(k_1 + 2k_0))\cdots));$$

for clarity, an example for $n = 7$ gives

$$\tilde{k} = k_6 + 2(k_5 + 2(k_4 + 2(k_3 + 2(k_2 + 2(k_1 + 2k_0))))).$$

This technique is used in the pseudo-code given below for computing the bit-reversed indices.

Pseudo-Code for Computing Bit Reversed Indices via Horner's Method

```
%% We want to bit reverse the indices 0 to N-1, where N=2^n
%% Note: implement multiply (divide) by two via shift left (right)

for k=0 to (N-1)                        % Loop over indices of signal samples
      temp = k                          % set current index value
        k_tilde=0                       % initialize reversed index to zero
        for i=0 to (n-2)                % Loop over the bits for Horner's Recursion
              if LSB(temp)=1            % check if LSB of temp is set
                 k_tilde=k_tilde+1      % if so, add 1 to k_tilde
              end if
              k_tilde=2*k_tilde         % make next higher power in  Horner's Recursion
              temp = temp/2             % shift right on temp
        end on i                        % End loop over bits
        if LSB(temp)=1                  % check if LSB of temp is set
            k_tilde=k_tilde + 1         % if so, add in last term of Horner's Recursion
        end if
         k_tilde_array(k)=k_tilde
end on k                                % End loop over indices of signal

%%%  k_tilde_array now holds the bit reversed indices
%%%  These BR'd indices are then used to reorder the signal samples for the FFT input array
```

Pseudo-Code Box 2: Compute Bit-Reversed Indices